Java基础

2020年3月11日 13:30

-
- JAVA基础

1. 序列化：将对象的内容流化成对象流再处理的机制
    1) 用途：保存到硬盘或在网络上以字节序列传送
    2) 如果不想让某域被序列化，可以加上transient关键字
    3) 静态域不会被序列化
2. 泛型（类型擦除）
    1) pecs原则：向下转换类型需要手动写
    2) 限定上界泛型<? extends E>：E类或E类的子孙类
        i. 参数是生产者，函数内只读，写时需要强制类型转换
    3) 限定下界泛型<? super E>：E类或E类的祖先类
        i. 参数是消费者，函数内只写，读时需要强制类型转换
    4) 任意泛型<?>
3. 匿名内部类和lambda函数式接口的实现对象
    1) 函数式接口：只有一个函数的接口，可用注解@FunctionalInterface
    2) Thread t = new Thread(new Runnable(){@Override public void run();});
    3) Thread t = new Thread(()->{run方法内容});
    4) 多线程：可返回结果的Callable接口和Future接口
    5) 线程池：ThreadPoolExecutor或ScheduledThreadPoolExecutor都可以执行Runnable接口和Callable接口
4. 反射
    1) 获取Class Object，即Class xx = xxx的xxx
        i. 对象.getClass()，对象.getSuperclass()
        ii. 类.class
        iii. Class.forName("包名.类名")
        iv. 基本数据类型的封装类型.TYPE
    2) 实例化对象，即Object xx = xxx 或 自定义类 xx = (强制向下类型转换) xxx
        i. 类对象.newInstance();
        ii. 类对象.getConstructor(new Class[]{}).newInstance(new Object[]{})
        iii. 类对象.getConstructor(Class<?>... parameterTypes).newInstance(Object... initargs)
    3) 获取变量域，Field[] xx = xxx
        i. 类对象.getFields()：当前类及超类的公开域
        ii. 类对象.getDeclaredFields();：当前类所有域
        iii. 类对象.getField(Stringname);：当前类及超类的指定公开域
        iv. 类对象.getDeclaredField(String name);：当前类指定域

4) 使用域
   i. 域对象.set(Object obj, Object value);：设置值
   ii. 域对象.get(Object obj);：获取值
   iii. 用反射更改private final元素：

```
// 获取String类中的value字段
Field valueFieldOfString = String.class.getDeclaredField("value");
// 改变value属性的访问权限
valueFieldOfString.setAccessible(true);
// 获取s对象上的value属性的值
char[] value = (char[]) valueFieldOfString.get(s);
// 改变value所引用的数组中的第5个字符
value[5] = '_';
```

来自 <https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247484891&idx=1&sn=65643683457bdd1d58cc9f4573d9788b&source=41>

5) 获取方法，即Method[] xx = xxx
   i. 类对象.getMethods()获得当前类及超类的public方法
   ii. 类对象.getDeclaredMethods()获得当前类的方法
   iii. 类对象.getMethod(String name, Class<?>... parameterTypes)获得前类以及超类指定的public方法
   iv. 类对象.getDeclaredMethod(String name, Class<?>... parameterTypes)获得当前类的指定方法
6) 调用方法：Method对象xx.invoke(对象, 参数列表)运行
7) ClassLoader实现动态加载修改类，资源隔离，热部署，代码加密

5. 控制反转Inversion of Control
   1) 依赖注入Dependency Injection
      i. 常见的实现控制反转的方法
      ii. 常由容器注入所依赖的对象
      iii. 常从构造函数、setter、接口注入
   2) 依赖倒置原则Dependency Inversion Principle
      i. 高层模块不应依赖低层模块，应依赖于抽象
      ii. 抽象不应依赖于细节，细节应依赖于抽象
   3) 控制反转是一种设计模式，实现了依赖倒置的设计原则
6. 代理模式/委托模式Proxy Pattern
   1) 一种面向接口间接访问对象的编程思想/设计模式
   2) 目的
      i. 扩展原有功能模块（原有方法的前置处理或后置处理）
      ii. 降低代码模块之间的耦合度
   3) 静态代理：所有调用目标对象的方法都改为调用代理对象的方法
      i. 因此对每个方法需要静态编码，较繁琐
   4) 动态代理：自动生成代理对象并调用对象方法
      i. 用下转型后的代理对象调用方法，会自动转发到InvocationHandler接口的方法invoke(proxy, method, args)

```
public interface myInterface {
   public int divide(int a, int b);
}
```

```
public class ProxyHandler implements myInterface { …… }
```

ii. 先实现InvocationHandler接口，生成其实现类的对象（可在其中前切片位置扩展调用前功能，再调用mehod.invoke(obj, args)，再扩展调用后功能）

```
public class ProxyHandler implements InvocationHandler {
   private Object obj;

   public ProxyHandler(Object o) { //构造方法依赖注入
       this.obj = o;
   }

   @Override
   public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
        //前切片
       int r = (int)method.invoke(obj, args);
        //后切片
       return r;
   }
}
```

iii. 用Proxy类静态方法newProxyInstance(classLoader, interface, handler)生成代理对象（再用依赖注入将被代理对象传给代理对象的成员obj）

```
InvocationHandler handler = new ProxyHandler(new Division());
myInterface proxy = (myInterface)Proxy.newProxyInstance(
       d.getClass().getClassLoader(),
       d.getClass().getInterfaces(),
       handler);
proxy.divide(a, b);
```

iv. 动态代理工厂：单独写一个工厂类，在其中的静态方法getProxy中调用Proxy.newInstance()，可配合匿名InvocationHandler类对象。另外将getProxy方法签名写成public static <T> T getProxy(final class<T>……并强转返回的代理对象类型可进一步简化调用难度（但需要@SuppressWarnings("unchecked")来忽视该方法的警告）

7. Aspect Oriented Programming面向切片编程AOP
   1) 基本概念：advice通知，joinPoint连接点，pointCut切入点，aspect切片，introduction引入，target目标，proxy代理，weaving织入
   2) 应用场景
      i. 在方法前后进行功能增强和扩展，如日志、安全权限、事务管理
      ii. 对方法执行顺序动态调整，Before，After，Around
   3) 优势
      i. 将横切业务和主业务逻辑剥离
      ii. 扩展功能不破坏原有业务逻辑
      iii. 专注主要业务逻辑
      iv. 代码复用
      v. 模块之间解藕
   4) 相关框架
      i. Spring的核心
         1) IoC
         2) AOP（jdk动态管理和Glib）
      ii. AspectJ（被整合进Spring的配置注解）
8. 小结

1) 反射：动态编程的基础
2) 面向接口：面向对象的精髓和生命力
3) IoC和DI设计模式：颠覆系统各层次间对象依赖关系
4) 动态处理模式：系统功能扩展解耦
5) AOP：新编程思想